MONITOR come strumento di programmazione concorrente

1. Struttura di un programma concorrente

Un programma concorrente si compone di processi concorrenti che usano come strumenti di sincronizzazione e comunicazione appunto monitors.

Sia i processi, sia i monitors (che non sono presenti in un normale programma Pascal) sono utilizzabili attraverso un costrutto di tipo: e' quindi necessaria una definizione del tipo corrispondente, poi una dichiarazione di una variabile di quel tipo. Una applicazione concorrente puo' quindi avere la seguente struttura:

```
program parallelo;
  < dichiarazione delle costanti >
  < dichiarazione dei tipi ; in particolare :
    type  nomemonitor =  monitor ;
        < struttura del monitor >  end;
        ....
    type nomeprocess =  process ( listaparametri formali);
        < codice del processo >   end;
        ....
  < dichiarazione delle variabili ; in particolare :
    M1   :  nomemonitor  ;  { si istanzia un monitor
                        del tipo definito }
    P1 ( listaparametrieffettivi ) :  nomeprocesso ;
       { viene istanziato un processo con parametri
         come specificato }
    .....

begin    { applicazione concorrente vuota }  end.
```

E' possibile una variante al precedente schema: la applicazione concorrente deve esplicitamente richiedere la inizializzazione sia dei processi che ha dichiarato sia dei monitor che saranno usati; cioe' :

```
begin   init  M1;
        init  P1; .... { altre inizializzazioni }
end.
```

Non ammettiamo che si dichiarino tipi o variabili concorrenti (processio monitor) all'interno di procedure in qualunque modo innestate.
I processi ed i monitor non hanno limitazioni nel loro tempo di vita, che e' pari alla intera pllicazione concorrente.

1.1 Processi

I processi sono quindi definiti attraverso il tipo corrispondente:

```
type  nomeprocesso = process ( p1: tipo1; p2: tipo2; ... );
        {i parametri del processo devono essere specificati al momento della dichiarazione }
  < dichiarazione delle costanti, tipi, variabili locali al processo >
begin
    <  codice del processo; e' possibile invocare anche procedure di monitor, che sono solo
successivamente istanziati : e.g. M1.operazione >
    .....
end.
```

1.2 Monitor

Il tipo monitor ha la seguente struttura:

```
type nomemonitor = monitor ;
  < costanti, tipi, variabili locali al monitor; il loro tempo di vita e' quello della intera applicazione
concorrente >
    procedure entry  nomeproc1 ( listaparametri ) ;
        < variabili locali, con tempo di vita limitato alla attivazione della procedura stessa >
    begin  < codice >  end;
    procedure entry  nomeproc2 .....
    .....
    function  entry  nomefunc1 ( listaparametri ) : tiporitorno;
        < sono ammesse anche funzioni invocabili dall'esterno >
    ....
    < si possono definire procedure e funzioni non visibili dall'esterno del monitor, ma utilizzabili
solo internamente >
begin
    { inizializzazione del monitor }
end.
```

2. Programmmazione concorrente

Una programmazione basata sugli strumenti presentati prevede quindi:

- **soggetti attivi**: i processi
- **oggetti condivisi**, racchiusi da monitor passivi.

Non sono consentite definizioni di processi all'interno di monitor e viceversa. Si evita cosi' la necessita' di definire una semantica per una gerarchia di processi e dell'innestamento dei monitor (vedi oltre).

Il costrutto monitor realizza il concetto di *tipo di dato astratto*:cioe' l'incapsulamento delle strutture dati insieme con le operazioni che agiscono sulle stesse da una parte (data abstraction); inoltre la possibilita' di non dover ridefinire il comportamento di piu' monitor che possono invece fare riferimento allo stesso tipo (typing).

Le variabili interne al monitor non sono accessibili direttamente dall'esterno e quindi sono protette in modo completo.

All'interno del monitor, per consentire una corretta sincronizzazione, sono utilizzabili altri tipi di dati astratti, opportunamente introdotti:

- variabili di tipo **condition**, insieme con le primitive che possono agire su queste:
  cond. wait;
  cond. signal;
  cond. queue;

< si adotta una notazione postfissa di uso, secondo le indicazioni di P.B. Hansen >

Le variabili condizione possono essere pensate come una sorta di coda. Le operazioni wait e signal hanno rispettivamente le funzioni di:

- sospensione del processo che la esegue, in coda alla variabile condition (accodamento FIFO).
- segnalazione del primo in coda alla variabile condition, se questo esiste; altrimenti nessun effetto. Tra il segnalante ed il segnalato e' necessario definire una politica di riattivazione opportuna.
- l'operazione queue permette di stabilire se la coda relativa e' piena o vuota.

Il vincolo fondamentale assicurato dal monitor e' la **MUTUA ESCLUSIONE**, ossia un solo processo per volta puo' essere in esecuzione (attivo) al suo interno. Altri processi possono essere sospesi e possono essere riattivati solo al termine di una procedure entry del monitor. Le richieste arrivate quando il monitor e' occupato sono sospese e possono essere riconsiderate .
Tutta la gestione delle richieste avviene con modalita' strettamente FIFO.

Le primitive **wait e signal** rappresentano un punto di rilascio di tale mutua esclusione (altrimenti, per esempio un processo sospeso non potrebbe piu' essere riattivato); esse comportano la possibilita' di lasciare entrare o riattivare altri processi.

Alla wait il processo attivo rilascia il monitor. Un processo in attesa puo' acquisirlo o il monitor viene dichiarato libero.

La implementazione della signal puo' anche essere sospensivaper il segnalante. La implementazione ipotizzata, per evitare il problema della alterazione delle condizioni che hanno portato alla segnalazione :
-sospende il processo segnalante (se esiste un segnalato)
-riattiva il primo processo in coda sulla condition.

Il segnalante viene sospeso su una coda prioritaria (gestita FIFO) detta **URGENTQUEUE**, i cui componenti hanno tutti priorita' rispetto ai processi ancora all'esterno del monitor ed in attesa dela sua liberazione. Questa soluzione implementativa consente una maggior potenza espressiva rispetto ad altre, in cui si vincola la posizione od il numero di invocazioni delle primitive sulle condition all'interno di una stessa procedure entry.

Il monitor garantisce, con la disciplina FIFO, un **trattamento giusto (o fair)** nella interazione tra processi, non penalizzando nessuno in alcun modo con una politica che invece sia sbilanciata a favore di alcuni processi.
Questo vale sia per i processi che sono stati sospesi in rispetto al vincolo della mutua esclusione, sia per i segnalanti sospesi sulla urgentqueue, sia per le code dei processi esplicitamente sospesi.

ADA nella soluzione di problemi concorrenti

Per le soluzioni di un problema di sincronizzazione con il rendez-vous esteso di ADA, possiamo procedere con alcune considerazioni:

- la sincronizzazione e' risolto attraverso una decomposizione del problema solo in termini di processi attivi: i task di ADA; non sono presenti oggetti condivisi;
- la **mutua esclusione** e' garantita automaticamente dalla presenza del processo servitore, che decide quando e come servire le richieste e come accedere al proprio stato interno di sincronizzazione;
-i **processi clienti** inviano le richieste al **processo gestore**; le richieste possono anche includere parametri in ingresso/uscita;
-il processo gestore in genere e' un processo senza fine che si apre su una selezione non-deterministica delle richieste che puo' accettare. L'accettazione delle richieste avviene in base allo stato del gestore stesso e **non sulla base dei parametri delle richieste stesse**: infatti, questi sono noti solo quando il rendez-vous e' gia' iniziato.
In caso al servitore sia necessaria una selezione che dipende dai parametri, le entry devono essere differenziate per fornirgli la conoscenza opportuna.
-Si noti che l'uso della select ammette **non-determinismo tra le entry da servire** se piu' di una e' aperta; una guardia e' aperta se la condizione di sincronizzazione e' verificata. Quindi, con questo schema di soluzione, e' piu' difficile tenere in conto di specifiche quali la 'fairness', ossia la giustizia in base all'ordine di accodamento.
D'altra parte, l'ordinamento in caso di sistema ad ambiente locale tende a perdere molto del significato che aveva nel concentrato: l'ordine di arrivo delle richieste al gestore puo' non essere conforme ai tempi in cui si sono manifestati gli eventi al gestore.
Il **non determinismo** e' qui introdotto per esprimere in modo unico un trattamento di casi diversi tra cui non ci interessa distinguere (non determinismo di scelta).
- Il **servizio di una entry** modifica opportunamente lo stato del gestore.

Uno **schema di soluzione** e' il seguente:

```
package body TuttaApplicazione is
type ...

task is clienteSpecifico is
end clienteSpecifico;

task body clienteSpecifico is
begin
    loop
        gestore.ACQ2;  { richiesta di procedura remota al GESTORE}
        <uso della risorsa>
        gestore.RIL5;
    end loop;                end clienteSpecifico;

-- piu' clienti con lo stesso comportamento
 task type cliente is end cliente;
```

```
task body cliente is
 begin
      loop
          gestore.ACQ1;
          <uso della risorsa>
          gestore.RIL1;
      end loop;
 end cliente;

task gestore is
  entry ACQ1;
  entry ACQ2;  ...
  entry RIL1;  ...
end gestore;
task body gestore is
            -- variabili
begin  -- inizializzazione variabili

loop
 select  -- when precede la guardia
   when <condizione sullo stato interno > ==>
   accept ACQ1  do
        < azioni di variazione dello stato >
     end ACQU1;
 or
   when <condizione > ==> accept ACQ2  do
      ...               end ACQUISsu;
 or
  -- alternativa sempre aperta
   accept RIL  do ...
     end  RIL;
  ...
 end select;
end loop;
end gestore;

a1, a2, ...  : cliente;
-- dichiarazione delle variabili task del tipo definito

end TuttaApplicazione;
```

Consideriamo le strutture dei processi:

LETTORI

```
type lettori= process;
begin
repeat
    lettscritt.LRICH;
        <leggi>
    lettscritt.LRIL;
until false;
end;
```

SCRITTORI

```
type scrittori= process;
begin
repeat
        lettscritt.SRICH;
        < scrivi >
        lettscritt.SRIL;
 until false;
end;
```

Usiamo due code separate, una per i processi lettori in attesa di accedere alla risorsa (codalett) ed una per i processi scrittori (codascritt); la risorsa stessa ha il proprio stato in un booleano occupato; e' inoltre necessario un contatore del numero di lettori in coda: nlettori.

1 versione

```
type rw= monitor
var codalett, codascritt: condition;
   occupato : boolean; nlettori: integer;

procedure entry LRICH;
begin     if ( occupato and nlettori = 0 ) then
             {  un lettore deve sospendersi se la risorsa e' occupata da uno scrittore }
             codalett.wait;
         occupato := true; nlettori := nlettori + 1;
         { segnalazione dei prossimi lettori in coda, se questi non sono tutti segnalati dal processo
                              scrittore che rilascia }
         codalett.signal
end;
procedure entry LRIL ;
begin     nlettori:= nlettori - 1;
         if nlettori = 0 then begin occupato := false;
                               codascritt. signal;
                          end;
end;
procedure entry SRICH;
begin     if occupato then { uno scrittore si sospende se la risorsa e' occupata }
             codascritt. wait;
          occupato := true;
end;
```

```
procedure entry SRIL;
begin     if codascritt. queue then codascritt. signal
              { se ci sono altri scrittori in coda, segnalali }
          else
              if codalett.queue then { ci sono lettori, segnalali }
                  codalett.signal
                              { se si vuole invece far segnalare dallo scrittore tutti i lettori, e'
                              necessarioeseguire
          invece della signal singola:
                    while codalett.queue do codalett.signal  }

              else occupato := false;
end;
```

L'applicazione parallela e' costituita da:

```
 program LettorieScrittori;

    { le dichiarazioni di tipo viste inserite qui }
     var lettscritt      :  rw;
        s1, ...        :  scrittore; {assieme ad altri processi}
        l1, ...         :  lettore  ;  { con altri }
 begin end.
```

Consideriamo adesso le modalita' per evitare starvation:

- si deve rendere impossibile l'acquisizione senza limite dei processi lettori, anche se la risorsa e' in possesso di un certo numero di lettori, se c'e' almeno uno scrittore in attesa

- si deve impedire agli scrittori di passarsi la risorsa tra loro, non considerando richieste di lettori gia' accodate.

```
type rw= monitor
var codalett, codascritt: condition;
    occupato : boolean; nlettori: integer;

procedure entry LRICH;
begin     if ( occupato and nlettori = 0 ) OR ( CODASCRITT.QUEUE)
          then
              {un lettore deve sospendersi se la risorsa e' occupata da uno scrittore, ma anche se c'e' almeno
                              uno scrittore in coda }
              codalett.wait;
          occupata := true; nlettori := nlettori + 1;
          codalett.signal
end;
procedure entry LRIL ;
```

```
begin     nlettori:= nlettori - 1;
        if nlettori = 0 then begin occupato := false;
                                   codascritt. signal,
                             end;
end;
```

```
procedure entry SRICH;
begin     if occupato then { uno scrittore si sospende se la        risorsae' occupata }
              codascritt. wait;
          occupata := true;
end;

procedure entry SRIL;
begin     if codalett. queue then codalett. signal
              { se ci sono lettori in coda, segnala il primo}
          else
              if codascritt. queue then
                  { c'e' uno scrittore, segnalalo }
                  codascritt.signal
              else occupato := false;
end;
```

Primo esercizio

Si consideri un incrocio formato da quattro strade destinato al traffico di sole automobili, come nella figura seguente, di direttrici Nord-Sud (NS) ed Est-Ovest (EO); le auto da qualunque direzione di provenienza possono o attraversare l'incrocio (procedendo diritto) o svoltare a sinistra.

1)Le due direzioni NS ed EO sono mutuamente esclusive: una macchina puo' passare in una direzione solo se l'incrocio non e' gia' impegnato da macchine nell'altra direzione.
Esiste una "direzione corrente" che e' determinata dalle macchine che impegnano l'incrocio. L'accesso all'incrocio puo' avvenire se e' sgombro oppure se la direzione della macchina coincide con quella corrente.
Ad esempio, se l'incrocio ha direzione di attraversamento EO, allora le auto in direzione NS devono essere bloccate (e saranno riattivate solo quando la direzione corrente sara' modificata, quando cioe' non ci sono piu' auto in attraversamento per EO, vedi figura).

2)Una volta che le auto hanno ottenuto l'accesso all'incrocio, esistono due tipi di transito:
a) attraversamento
b) svolta a sinistra.

I due tipi sono mutuamente esclusivi e viene data priorita' all'attraversamento.
Ad esempio, un'auto che intende svoltare a sinistra puo' farlo se non ci sono auto che stanno completando l'attraversamento o se non ci sono auto sospese che hanno manifestato l'intenzione di attraversare l'incrocio (la sospensione deriva dalla mutua esclusione dei due tipi di transito).

3)Nella fase di rilascio, la direzione viene cambiata solo quando tutte le macchine che hanno avuto accesso all'incrocio hanno completato il transito.

Si costruisca un monitor che fornisca le procedure corrette (di acquisizione e rilascio) e si definiscano i processi per simulare un comportamento in accordo alle specifiche sopra definite.

Si assicuri che non si verifichi la situazione seguente: l'incrocio e' sgombro e ci sono auto in attesa solo nella direzione opposta alla corrente.

Facoltativamente

Preso atto delle situazioni di starvation rese possibili dalla politica descritta precedentemente, se ne realizzi una diversa che le prevenga. In particolare non si consenta un passaggio indiscriminato alle auto in direzione corrente, ma si obblighi un cambiamento della direzione corrente dopo almeno un prefissato numero di passaggi di auto (p.e. NPASSAGGI), se ci sono veicoli gia' in coda per accedere all'incrocio nella direzione opposta.

Consideriamo il problema dell'incrocio con le specifiche date qui sopra: una struttura di soluzione e' la seguente.

```
program  incrocioConAutoCheAttraversanoESvoltano;
type  d = ( NS, EO );
     svolta = ( diritto, sinistra);

type  incrocio = monitor;

var  dir : d ;  { direzione corrente dell'incrocio }
    cont : integer; { numero di auto che hanno impegnato gia'          l'incrocio e non hanno completato il
transito}
          contint: array [svolta] of integer;
                { numero di auto che sono presenti per svoltare o andare diritto }
          coda: condition; {per le auto in direzione non corrente}
          codeint : array [svolta] of condition;
                { per la sospensione nel transito}

procedure entry entraDiritto (direz: d);
begin
    if (dir <> direz and cont <> 0 ) then  coda.wait;
              { la sospensione e' fatta solo se l'incrocio e' gia'        impegnato in direzione opposta }
          if cont = 0 then dir := direz;
          cont := cont + 1;
          if contint [sinistra] <> 0 then
              { stanno passando auto che svoltano }
              codeint [diritto]. wait;
          contint [diritto] := contint [diritto] + 1;
          end;

procedure entry entraSinistra (direz: d);
begin
    if (dir <> direz and cont <> 0 ) then  coda.wait;
              { la sospensione e' fatta solo se l'incrocio e' gia'        impegnato in direzione opposta }
          if cont = 0 then dir := direz;
          cont := cont + 1;
          if (contint [diritto] <> 0 or codeint[diritto].queue) then
              { stanno passando auto che attraversano o ce ne sono         gia' bloccate}
```

```
            codeint [sinistra]. wait;
        contint [sinistra] := contint [sinistra] + 1;
end;
```

```
procedure entry rilasciaDiritto;
begin
        cont := cont - 1;
        contint [diritto] := contint [diritto]- 1;
        if contint [diritto] = 0 then  begin
            if codeint[sinistra].queue  then{ si segnalano nel caso che sia terminato il transito le
                auto nella stessa direzione}
                        while codeint[sinistra]. queue  do
                            codeint [sinistra].signal
                              end
            else { segnalazione sospesi nella direzione opposta }
                while coda.queuedo coda.signal;
        { in caso che contint del verso di svolta corrente sia <> 0 allora non si esegue nessuna azione}
end;

procedure entry rilasciaSinistra;
begin
        cont := cont - 1;
        contint [sinistra] := contint[sinistra]- 1;
        if contint [sinistra] = 0 then
            if codeint[diritto].queue  then{ si segnalano nel caso che sia terminato il transito le
                auto nella stessa direzione}
                        while codeint[diritto]. queue  do
                            codeint [diritto]. signal
            else { segnalazione sospesi nella direzione opposta }
                while coda.queuedo coda.signal;
        { in caso che contint del verso di svolta corrente  sia <>0 allora non si esegue nessuna azione}
end;

begin { monitor }
 cont := contint[diritto] := contint [sinistra] := 0;
 dir := EO;
end;

type  autoDiritta  (direz : d) = process;
begin
    while true do  begin
       crocevia. entraDiritto (direz);
        < transito >
        crocevia. rilasciaDiritto;
    end;
end;

type  autoSinistra  ( direz : d) = process;
begin
    while true do begin
```

```
        crocevia. entraSinistra (direz);
         < transito >
         crocevia. rilasciaSinistra;
    end;
end;

var  crocevia : incrocio;
    a1 (NS), a2 (EO) , a3 (NS),..., an (EO) : autoDiritta;
    b1 (NS), b2 (EO) , b3 (NS),..., bn (EO) : autoSinistra;
```

begin end. Nel caso si voglia evitare le starvation, si puo' per prima cosa fare in modo che la direzione dell'incrocio non possa essere mantenuta per piu' di NPASSAGGI di auto.

```
program incrocioSenzaStarvation;

type  d = (NS, EO);
     svolta = (diritto, sinistra);
type  incrocio = monitor ;

var  dir : direz;
    cont: integer;
    npass : integer;  {numero di auto gia' passate per la direzione corrente}
          coda: array [ d ] of condition;
        {  per le auto in direzione non corrente}

    contint :  array [ svolta ] of integer;
        { si mantengono qui contatori delle auto sospese in              entrambe le direzioni }
         codeint : array [svolta] of condition;
             { per la sospensione nel transito}



procedure entry entra (direz: d; s : svolta);
begin
    while ( (dir <> direz and cont <> 0 ) or
        ( npass = NPASSAGGI) )     do  coda[direz].wait;
             { la sospensione e' fatta se l'incrocio e' gia'           impegnato in direzione opposta o quando
         sono gia'                    transitate NPASSAGGI auto}
    if cont = 0  then dir := direz;
          cont := cont + 1; npass := npass +1;
          contint [s] := contint [s] + 1;

    if s = diritto then  begin
            if contint [sinistra] <> 0 then
              { stanno passando auto che svoltano }
              codeint [s]. wait;
```

```
        else  if (contint [diritto] <> 0 or codeint[diritto].queue)
                    {ci sono auto che attraversano o che vogliono     attraversare}
              then  codeint [s].wait;
    end;
end;
```

```
procedure entry rilascia (direz: d; s : svolta);
begin
        cont := cont - 1;
        contint [s] := contint [s]- 1;
        if contint [s]  = 0 then
            if codeint[opposite(s)].queue  then{ si segnalano nel caso che sia terminato il
                        transito le auto nella stessa direzione}
        while (codeint[opposite(s)]. queue
                              codeint [opposite(s)].signal
       else { segnalazione sospesi nella direzione opposta } begin
          npass := 0;
               while ( coda[ opposite(direz)] . queueand
                npass <> NPASSAGGI )  do
              coda[ opposite (direz)].signal;
          end;
        { in caso che contint sia <>0 allora non si esegue nessuna azione}
end;

begin
    cont:= contint [diritto] := contint [sinistra] := 0;
    npass := 0;
    dir := NS;
end;

type  macchina ( direz : d ; s : svolta) = process;
begin
  while true do begin
    crocevia. entra (direz, s);
    < transita>
    crocevia. rilascia (direz, s);
  end;
end;

var crocevia : incrocio;
   a1 (EO, diritto), ...  : macchina;
   b1 (EO, sinistra), ... : macchina;
   c1 (NS, diritto), ...  : macchina;
   d1 (NS, sinistra), ... : macchina;
begin end.
```

Problema dei filosofi e del piatto di spaghetti

Si considerino cinque filosofi che passano il loro tempo pensando e mangiando. Ogni filosofo ha il proprio posto in una tavola circolare ed ha davanti un piatto di spaghetti.
Per mangiare gli spaghetti un filosofo necessita di due forchette. Ce ne sono solo cinque (F0, F1, F2, F3, F4) posta ciascuna tra due filosofi.
Ogni filosofo puo' prendere solo le due forchette poste immediatamente alla sua destra ed alla sua sinistra. Ovviamente filosofi adiacenti non possono mangiare contemporaneamente ed al piu' due filosofi possono avere le forchette disponibili per mangiare.

I filosofi si comportano allo stesso modo: ogni filosofo si comporta come un processo che utilizza le procedure di un monitor per una corretta acquisizione delle risorse: Acquisizione, Rilascio.
Lo schema del processo e' quindi:

```
repeat
    < acquisizione delle forchette >
    mangia gli spaghetti
    < rilascio >
    pensa tranquillo
until false;
```

In particolare, il problema e' di evitare situazioni di deadlock, in relazione alla acquisizione delle forchette.

Si scriva il monitor introducendo le seguenti strutture dati:

forks[i] i= 0..4
contiene il numero di forchette disponibili per un processo filosofo. Il numero puo' valere 0, 1, 2.
La condizione di sincronizzazione e' fork[i]= 2 per il processo i-esimo.

ready[i] i= 0..4
array di variabili condizione su cui il processo i-esimo si sospende quando la condizione di sincronizzazione non e' verificata.

left(i), right(i) i= 0..4
funzioni che forniscono l'indice del vicino di sinistra e di destra per il filosofo i-esimo: in particolare, right(4)= 0 e left(0)= 4.

Consideriamo direttamente l'applicazione parallela costituita da un processo per ogni filosofo e da un unico monitor per regolare l'accesso alle forchette.

```
program filosofiedununicopiattodispaghetti;

type numerof = 0 .. 4;

type filosofo = process ( i : numerof );
begin
    repeat    tavolo. acquire ( i );
              < mangia >
              tavolo. release ( i ),
    until false;
end;

type table = monitor ;

var  forks : array [ numerof ] of 0..2;
        { struttura per tenere traccia delle forchette              disponibili per un filosofo ad un certo istante
}
    ready : array [ numerof ] of condition ;
        {   batteria di variabili condizione per la sospensione             dei filosofi }
    i     : numerof  ; {  variabile temporanea }

function left ( i : numerof ) : numerof ;
function right ( i : numerof ) : numerof ;
        {   funzioni con ovvio significato }

procedure entry acquire ( i : numerof );
begin
    if forks [i] <> 2 then ready [i].wait;
        { alla riattivazione sono decrementate le              disponibilita' dei vicini }
    forks [ left [i] ] := forks [ left [i] ] - 1;
    forks [ right [i] ] := forks [ right [i] ] -1;
end;

procedure entry release (i : numerof );
begin
    forks [ left [i] ] := forks [ left [i] ] + 1;
    forks [ right [i] ] := forks [ right [i] ] + 1;
    if forks [ left [i] ] = 2 then ready [ left [i] ]. signal;
    if forks [ right [i] ] = 2 then ready [ right [i] ]. signal;
        { nel caso il vicino non sia sospeso le segnalazioni non hanno alcun effetto  }
end;

begin
```

```
    for i := 0 to 4 do forks[i] := 2;
end;

    { dichiarazioni globali di monitor e processi }
var      tavolo : table ;
         f0   :  filosofo (0);
         ...
         f4   :  filosofo (4);
begin
end.
```

Possiamo considerare una soluzione diversa in cui un processo in fase di rilascio anziche' controllare la condizioni per i processi da segnalare, li riattiva in modo incondizionato: e' allora necessario che i segnalati provvedano al controllo della possibilita' di procedere od eventualmente risospendersi, questa modalita' viene spesso detta a retest (da parte dei processi riattivati) ed e' particolarmente adatta o a volte l'unica praticabile quando le condizioni di prosecuzione sono note ai soli processi che si sono sospesi.
Qui, le modifiche sarebbero:

in acquire anziche' una alternativa, una ripetizione:

```
    while forks [i] <> 2 do ready [i]. signal ;
```

in release due segnalazioni incondizionate:

```
    forks [ left [i] ]. signal ;
    forks [ right [i] ]. signal ;
```

Accanto a questa soluzione del tutto centralizzata, in cui cioe' un unico monitor previene ogni possibilita' di deadlock, sono possibili soluzioni piu' parallele:
per esempio, un monitor per ogni forchetta.

Si noti che in questo caso e' necessario evitare il problema del deadlock con una opportuna politica a livello di processi.
Una politica puo' essere la seguente:

- per un filosofo di posto pari, la prima forchetta che viene acquisita e' la forchetta di posto pari corrispondente; solo una volta acquisita questa, la forchetta di posto dispari puo' essere acceduta.
- per un filosofo di posto dispari, prima viene acquisita la forchetta di posto pari corrispondente, e solo successivamente, quella di posto dispari.

In questo modo le situazioni di deadlock sono evitate; infatti:

- un conflitto su una forchetta pari lascia un processo in grado di proseguire (il vincitore) e l'altro senza nessuna risorsa (che quindi non puo' rappresentare il possessore di una risorsa necessaria ad un altro).

- un conflitto su una forchetta dispari blocca si' un processo che ha gia' acquisito una risorsa, ma il vincitore della contesa ha gia' tutte le risorse necessarie per la prosecuzione ed il termine del proprio pasto.

La circolarita' tipica di situazioni di deadlock e' quindi completamente evitata.

```
program filosoficonmonitorperogniforchetta;
type numerof = 0..4;

type filosofo = process ( i : numerof );
begin
    forks [ fork1 (i) ] . acquire;
    forks [ fork2 (i) ] . acquire;
    < mangia >
    forks [ fork1 (i) ] . release;
    forks [ fork2 (i) ] . release;
end; type forktype = monitor ;
var  taken : boolean ;  queue : condition ;
procedure entry acquire ;
begin     if taken then queue.wait; taken := true;  end ;
procedure entry release ;
begin     taken := false ; queue. signal ;  end;

begin     taken := false ; end;

function  fork1 ( i : numerof ): numerof ;
begin     if i = i / 2 * 2  { cioe' pari } then fork1 := i
        else          fork1 := i (+ mod 4) 1;
end;
function  fork21 ( i : numerof ): numerof ;
begin     if i <> i / 2 * 2  { cioe' pari } then fork2 := i
        else          fork2 := i (+ mod 4) 1;
end;

var  fork : array [ numerof ] of forktype ;
    fo : filosofo (0);
    ...
    f4 : filosofo (4);

begin end.
```

Si pensi ad una soluzione con un monitor per ogni forchetta, in cui il deadlock venga evitato con l'utilizzo di un ulteriore monitor arbitro che e' incaricato di gestire la stanza in cui si trova il piatto di spaghetti: suo compito e' quello di non lasciare accedere alla stanza piu' di quattro filosofi alla volta.

Schematizziamo per prima cosa una soluzione che racchiude tutte le procedure a disposizione in un unico monitor, determinando quindi dei vincoli rispetto al parallelismo possibile.

```
program rotatoriaconcapacitamassima;

const N = numerorami ; { per esempio 6 }

type  rami = 1 .. N ;

type  rotonda = monitor;
const NMAX = ..; { numero massimo di auto possibili }
var       cont : 0 .. NMAX ;  j : rami;
        enter , daiprec : array [ rami ] of condition;
          {la batteria di condition enter consente la sospensione delle auto all'esterno della rotonda,
                        quando questa e' piena, la daiprec consente la sospensione dei processi gia' in
                        rotazione rispetto a quelli in ingresso nella rotonda che devono avere la
                        precedenza }
        content : array [rami ] of integer;
            {contatore delle auto gia' entrate nel ramo, ma non ancora transitate ed entrate in rotonda }

        j    :  rami  { variabile di appoggio }

procedure entry INGRESSO (i : rami );
begin
    if cont = NMAX then enter [i].wait;
        { se la rotonda e' piena, sospensione}
    cont + := 1;  content [i] + := 1;
end;

procedure entry RUOTA  (i,o: rami);
var j : rami;
begin
    content [i] - := 1;
    if content [i]= 0 then
        while daiprec[i].queue do daiprec[i].signal;
        { vengono segnalati tutti i processi che  si sono sospesi ed hanno dato la precedenza alle
                macchine in transito }
    for j := succ(i) to prec(o) do
        { per ogni ramo intermedio della rotonda si da la precedenza ai rami attraversati}
        while cont< NMAX and content[j] <>= 0  do
                daiprec [j]. wait;
end;

function cequalcunoincoda : boolean;
begin
      cequalcunoincoda := false;
```

```
      for j:= 1 to N   do
        if enter [j].queue then cequalcunoincoda := true;
end;
{ le funzioni succ e pred hanno l'ovvio significato di somma e differenza modulo N }
```

```
procedure entry ESCI ( o: rami );
var j: rami ;
begin     cont - := 1;
         while cont < NMAX and  cequalcunoincoda do
           for j := 1 to Ndo
              while cont < NMAX and enter[j].queue do
                  enter [j].signal;
end;

begin
    cont := 0; for j in rami do content [j] := 0;
end;

type  auto = process;
begin
    repeat
        ROT. INGRESSO ( i );
        ROT. RUOTA ( i, o);
        ROT. ESCI ( o );
    until false;
end;

var ROT : rotonda;
   pr,.....  : auto;

begin end.
```

La starvation non e' ovviamente bandita: infatti la riattivazione dei processi esterni bloccati e' sempre nello stesso ordine: quindi un processo sui rami bassi e' sempre favorito. Si puo' pensare a politiche piu' giuste e corrette (spesso indicate con il termine inglese di FAIR e FAIRNESS):

```
 j := succ ( o );
 while  cont < NMAX and cequalcunoincoda   do
    begin enter [j]. signal ;
         j := succ (j);
    end;
```

La riattivazione avviene sempre a partire dal ramo successivo a quello di uscita; o, per esempio a quello di ingresso, o alternatamente. Si possono pensare altre politiche, riattivando a partire da un ramo, contenuto come indicazione in una variabile che viene aggiornata ad ogni riattivazione e che scorre circolarmente sui rami della rotatoria.

Per quanto riguarda il maggiore parallelismo, si noti che:

1) si puo' pensare ad una struttura con un monitor per ogni ramo, garantendo quindi che un accesso ad un ramo non comporti il blocco di tutti gli altri rami.
Ogni monitor mette a disposizione tre procedure : entra, esci e ruota.

2)E' necessario considerare anche un monitor come arbitro: deve infatti esistere un gestore globale che sia capace di tenere conto delle condizioni globali (cioe' il numero totale di auto presenti nella rotonda).

3) I monitor pero' sono in stretta interrelazione tra loro:

i monitor di ramo devono poter chiamare delle procedure dell'arbitro, per conoscere la situazione della rotonda e per poterla variare (incrementarla/ decrementarla).
Questo non introdurrebbe scorrettezze, in quanto l'arbitro non invocherebbe i monitor di ramo.

i monitor di ramo devono anche segnalarsi tra di loro nel caso di segnalazione globale di auto in attesa di entrare: cio' reintroduce il problema del deadlock.

Problema dell'ascensore

Definiamo la politica dell' ascensore nel modo piu' semplice possibile per evitare la starvation: l'ascensore spazzola dal piano superiore al piano inferiore e viceversa, anche se non sono presenti richieste per i piani verso cui si sta muovendo.
Politiche piu' sofisticate od efficienti sono possibili.

Consideriamo un processo per l'ascensore ed un processo per ogni singolo utente. Il monitor disciplina gli accessi alla struttura che rappresenta lo stato corrente della risorsa ascensore, cioe' il piano e la direzione di movimento.

```
program ascensoreSenzaLimitiDiCapacita;
const N = ..;
type piano = 1 .. N;
    dir  = ( su, giu );

type ascensore = monitor ;
var  pianocor : piano;
    dircor   : dir;
    csu, cgiu, carr : array [ piano ] of condition;
{definiamo tra condizioni per ogni piano:
          le prime due sono per l'accesso all'ascensore nei due versi;
          l'altra per l'arrivo al piano.
          Le condizioni consentono di sincronizzazre l'utente con il movimento dell'ascensore }

procedure entry mov ( part : piano; arrivo : piano);
var  mydir : dir;
begin
    if arrivo > part then mydir := su else mydir := giu;
    if arrivo <> part then
    begin
    { accodamento per l'ingresso all'ascensore }
        if (part <> pianocor) or (mydir <> dircor) then
            if mydir = su then csu  [part]. wait
            else               cgiu [part].wait;
    { l' utente deve essere accodato al piano di arrivo }
        carr [ arrivo]. wait;
    end;
end;

procedure entry spostaAscensore;
begin
    if dircor = su then
        while csu [ pianocor] .queue do csu [pianocor].signal;
            { per ogni piano segnala tutti i processi sospesi in coda in attesa di entrare nell'ascensore, se la
                        direzione e' corretta }
```

```
    else while cgiu [ pianocor] .queue do cgiu [pianocor].signal;

    while carr [pianocor] . queue  do carr [pianocor]. signal;
            { per ogni piano segnala tutti i processi sospesi in coda in attesa di arrivare }
    if dircor=su then
        if pianocor <> N  then pianocor + :=1
            else            dircor := giu
    else if pianocor <> 1  then pianocor - := 1
            else            dircor := su ;
end;
begin dircor := su; pianocor := 1;
end;
type utente = process ;
var piano1, piano2 : piano;
begin
    repeat
        < scegli direzioni >
        elevator. mov ( piano1, piano2);
    until false;
end;

type asc = process;
begin repeat
        elevator. spostaAscensore;
    until false;
end;

var  elevator : ascensore;
    muoviasc : asc;
    u1, ...  : utente;

begin .. end.
```

{ Si possono prevedere gestioni ottimizzate del movimento dell'ascensore: per esempio di 'spazzolare' fermandosi solo sui piani per i quali c'e' almeno una richiesta.

Questa ottimizzazione della politica di gestione rimane confinata all'interno della procedura spostaAscensore }